MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.01—Introduction to EECS I
Spring Semester, 2008

Assignment 11, Issued: Thursday, April 24

Overview of this week's work

No lecture or software lab

Before the start of your design lab on Apr 24 or 25

- Read the week 10/11 class notes.
- Do the on-line tutor problems in section 11.1.

In design lab

- Do the nano-quiz.
- Work through the design lab.

By the beginning of your next software lab on Apr 29 or 30

- Do the on-line tutor problems in section 11.2.
- Submit written solutions to questions 6, 7, 8, 9, 11, and 12. All written work must conform to the homework guidelines on the web page.

Do `athrun 6.01 update` to get a directory `lab11` that contains the relevant files. In order to do this entire lab, it will also be important to have the latest version of SoaR. Athena machines and lab laptops will update automatically; to update your laptop, please download a new version of SoaRfrom the 6.01 software page.

Planning in the real world

This week, we'll assume that the robot can know exactly where it is in the world, and plan how to get from there to a goal. Generally speaking, this is not a very good assumption, and we'll spend the next two weeks trying to see how to get the robot to estimate its position using a map. But this is a fine place to start our study of robot planning.

We will do one thing this week that doesn't seem strictly necessary, but will be an important part of our software structure as we move forward: we are going to design our software so that the robot might, in fact, change its idea of where it is in the world as it is executing its plan to get to the goal. This is very likely to happen if it is using a map to localize itself (you've probably all had the experience of deciding you weren't where you thought you were as you were navigating through a strange city). This week, the only way it can happen is if, in the simulator, a malicious person drags the robot to another part of the world as it is driving around.

If we're going to use search algorithms to plan paths for the robot, the biggest question, as always, is how to represent the real world problem in our formal model. We need to define a search problem, by specifying the state space, successor function, goal test, and initial state. The choices of the state space and successor function typically have to be made jointly: we need to pick a discrete set of states of the world and an accompanying set of actions that can move between those states.

Here is one candidate abstract state-space formulation:

states: Let the states be a set of grid squares described by the robot's indices, `i` and `j` into a two-dimensional grid of square cells, overlaid on the original `x, y` coordinate space of the robot. In this abstraction, the planner won't care about the orientation of the robot; it will think of the robot as moving from grid square to grid square without worrying about its heading, or about its detailed position within that square.

actions: The robot's actions will be to move North, South, East, or West from the current grid square, by one square, unless such a move would take it to a square that isn't free (that is, that could possibly cause the robot to collide with an obstacle). The successor function returns a list of states that result from all actions that would not cause a collision. (We will sometimes call these abstract actions "macro-actions".

goal test: The goal test can be any Boolean function on the location grids. This means that we can specify that the robot end up in a particular grid square, or any of a set of squares (somewhere in the top row, for instance). We cannot ask the robot to move to a particular `x, y` coordinate at a finer granularity than our grid, to stop with a particular orientation, or to finish with a full battery charge.

initial state: The initial state can be any single grid square.

The planning part of this is relatively straightforward. The harder part of making this work is building up the framework for executing the plans once we have them. We still have to be able to control the robot's wheel velocities in a way that will allow it to move reliably from square to square, to support the abstraction we've designed.

The `HGBrain` works roughly as follows:

- It finds its current actual pose.

- It converts the current world location to grid coordinates and plans a path from there to a goal grid location.
- It executes the first "macro-action" in the plan, by driving approximately one grid square to the north, south, east, or west.
- Once it finishes the macro-action, it plans again (just in case someone has kidnapped it[1]), and executes the first macro-action in the new plan.
- Once it is near the goal location, it quits driving (though it keeps rechecking its location, again, just in case of kidnapping).

Try it out!

Question 1: Read the handout up to this point. (And be sure to read the text in the handout as you go.)

Question 2: Try out the planner and see how it works.

- Start Soar
- Pick **Simulator**, then `she.py` (which specifies the "She World" (named after three LAs who defined it) for the simulator).
- Pick **Brain**, then `HGBrain.py` (from the `6.01/lab11/`) directory.
- You'll see a new window, with a picture of the environment drawn as a grid of squares, with the occupied ones drawn in black and the free ones in white. Furthermore, it shows the robot's current plan. The green square is centered on the robot's actual current pose. The gold square is the goal, and the blue squares show the rest of the steps in the plan.
- Click start. The robot will drive through the world, with the plan redrawing each time a subgoal is achieved, until it gets to the goal.
- While the robot is moving, drag the robot somewhere else. See what happens.

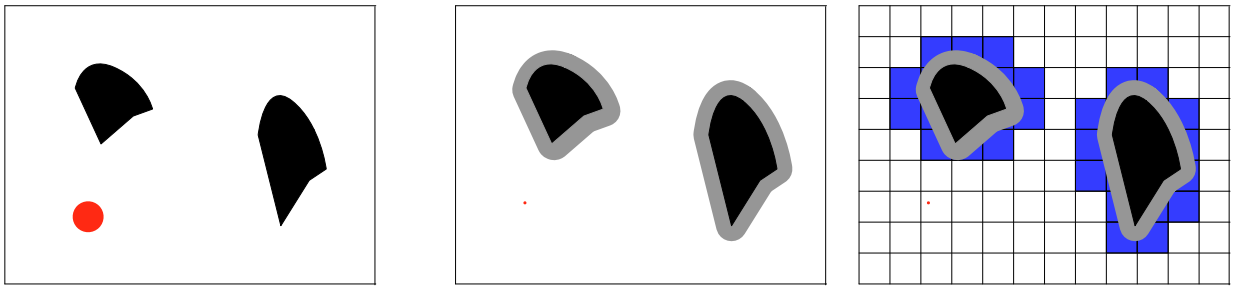
GridWorld.py

In the next couple of sections, we will walk through some of the code that makes this work, starting with `GridWorld.py`. This file defines two classes. The `Map` class is used to specify the locations of obstacles in the world, and the `GridWorld` class specifies a state-machine model of a robot moving on a grid corresponding to a map.

A `Map` object just stores a set of "boxes" that describe where the obstacles in the world are. All of our obstacles are rectangles, represented by the coordinates of two diagonal corners.

The `D2GridWorldFromMap` class is a subclass of `PrimitiveSM` (a primitive state machine) representing an abstract space of legal poses of the robot and motions between them. An abstract state is described by the robot's indices, `i` and `j` into a two-dimensional grid of square cells, overlaid on the original `x, y` space. A grid cell is *free* if the robot's center can be anywhere in that square of locations, and can be in any orientation, and not collide with any obstacle. We will treat the robot as if it is round, with radius `robotWidth`, which simplifies matters, but makes our planning overly conservative. (If your robot is round, a simple way of dealing with obstacles is to "grow"

[1]There is, believe it or not, a whole technical literature on the "kidnapped robot" problem, in which the the robot is moved to some completely unpredictable location from where it currently is. It will be even more interesting to deal with that problem two weeks from now.



(a) Original obstacles with round robot

(b) Obstacles grown by robot's original radius with robot of zero radius.

(c) Grid with cells marked occupied if they contain *any* position for which the robot would collide with an obstacle.

Figure 1: Construction of a grid map in two dimensions for a round robot

the obstacles by the radius of the robot, and then think of the robot as a point, moving through the space of fatter obstacles. Convince yourself that this is true.)

Figure 1(a) shows a simple world with two obstacles and a round robot. Growing the extent of each obstacle by the radius of the robot gives us the picture in figure 1(b), in which we can now think of the robot as a point, and any place in the `x,y` plane where it is not overlapping with the grown obstacles as being free. Finally, in figure 1(c) we show a grid superimposed on the world, and any cell that contains a position for which the robot would be in collision with an obstacle marked as occupied.

The initializer for a `D2GridWorldFromMap` takes as input an object of the `Map` class, which specifies where the obstacles are, the minima and maxima of the `x` and `y` coordinates, and a granularity, `gridN`. The grid will have `gridN` cells in each dimension.

```
class D2GridWorldFromMap (D2GridWorld):
    robotWidth = 0.2
    def __init__(self, theMap, xmin, xmax, ymin, ymax, gridN):
```

Here are the most important methods of the `D2GridWorldFromMap` class. See the code file for the definitions of these, and a bunch of other useful methods, if you're interested. Much of the work here is in translating between poses in real world coordinates, and those in "index" coordinates. Indices map into the stored array. So, for instance, if our `x` coordinate went from -10 to +10, and `gridN` were 40, then index 20 would be associated with `x` coordinate of 0.

```
    # specify the possible inputs to the grid state machine;  these are
    # the robot's action specifications that move it
    # west, north, east, south, or stay
    # put.  All inputs are possible in every state, though if the
    # robot tries to drive through an obstacle, it will just stay in its
    # original pose
    def legalInputs(s):
        return [(0, 0), (-1, 0), (+1, 0), (0, +1), (0, -1)]

    # Is the cell with indices i,j free?  Returns a boolean
    def free(self, (i,j)):

    # Specify the state transition function.  Takes a state which is a pair of
    # indices and an action, which is also a pair of indices, and returns
```

```
    # the resulting state, which is the sum of those indices if that
    # resulting state is free and returns the original state otherwise
    def transitionFn((ix, iy), (dx, dy)):
        newix = ix + dx
        newiy = iy + dy
        if self.free((newix, newiy)):
            return (newix, newiy)
        else:
            return (ix, iy)

    # Convert (x,y) world coordinates to (i,j) indices
    def poseToIndices(self, (x,y)):
    # Convert (i,j) indices to (x, y) world coordinates
    def indicesToPose(self, (ix,iy)):
    # Draw the grid map in a window, with occupied squares black and
    # free squares white
    def drawWorld(self, window):
    # Draw the robot in a location on the grid map
    def drawState(self, indices, window, color):
```

In addition, the file contains definitions of lists of boxes corresponding to the obstacles in many of the worlds in the simulator.

Question 3: Let's try out the code; load the `GridWorld.py` file in Idle.

a. A `Map` is created from a list of boxes, such as `sheBoxes`. What do:

```
Map(sheBoxes).boxes
Map(sheBoxes).segs
```

contain? What do these represent?

b. The `D2GridWorldFromMap` class creates a state machine that we can use to plan paths in a world while avoiding obstacles.

```
gw = D2GridWorldFromMap(Map(sheBoxes), 0, 4, 0, 4, 20)
```

This creates a gridworld which is 4 meters x 4 meters divided into 20x20 grid squares. `gw` is actually a `PrimitiveSM` with additional methods to map between robot coordinates and grid indices. What are the values of the following expressions:

```
gw.poseToIndices((3.5, 0.5))
gw.indicesToPose((18,3))
gw.free((18,3))
gw.free((19,3))
gw.free((18,4))
gw.transitionFunction((18,3),(1,0))
gw.transitionFunction((18,3),(0,1))
```

Explain the results.

c. Since `gw` is a `PrimitiveSM` we have seen that we can use a `Planner` instance to plan paths using the successor function defined from the state machine:

```
from Planner import Planner
pl = Planner(gw)
```

Use `pl` to plan a path from `(18,3)` to `(18,5)`. Try it again with depth first search. Explain the results. Remember that the planner has a variable `sm`, which is the state machine describing the world it is working in, and that the arguments to `plan` method are:

- `initialState`: Taken by default from `sm.currentState` (note that this variable won't exist unless you have already called the `start` method of the `sm`; that will be useful later on, but not now).
- `goalTest`: Taken by default to be `sm.terminationFunction`, but you can supply your own. Should take a state as input and return a Boolean.
- `depthFirst`: Taken by default to be `False`, meaning that, by default, the search is breadth first.

XYDriver.py

The `XYDriver` is in charge of making the robot drive straight from one real-world location to another. We will use it to drive from one grid cell to the next, assuming that the space between them is free. This should be a fairly familiar task, from week 4 and from the midterm.

We start by defining a basic class `XYDriver`, which is a terminating state machine, in the sense that it supplies `start`, `step`, `currentOutput` and `done` methods. It is given a relative goal location in world coordinates, and when the `start` method is called, it computes an absolute goal location,

which is the relative goal with respect to its current location. Then, on subsequent `step`s, it will try to drive straight from its current location to the absolute goal location. It expects sensor inputs (sonar readings and pose) as inputs, and generates robot velocities (pairs of rotational and forward velocities) as outputs.

The arguments to the initializer are:

- `deltaXY`: a relative goal for the robot in *world* (x, y) coordinates (not indices)
- `distEps`: a distance specifying how close the robot should come, in global coordinates, to the goal, before terminating.

```
class XYDriver:
    forwardGain = 1.5
    rotationGain = 1.5
    angleEps = 0.05

    def __init__(self, deltaXY, distEps):
        self.deltaXY = deltaXY
        self.distEps = distEps
```

The behavior first rotates until it is aligned towards the goal location (within `angleEps`) and then moves straight towards the goal location. It uses the gains to choose velocities as a function of the current distance from the goal. That is, it uses a simple proportional controller of the type we have seen many times before.

The basic `XYDriver` behavior has one major flaw: if something is in its way, the robot will run right into it. Here, we define a new class, `XYDriverOA`, (the "OA" stands for *obstacle avoidance*), which looks at the sonar readings and terminates the behavior if any reading is shorter than some threshold. It is a subclass of `XYDriver`, so it just overrides the methods it needs to change.

The initializer is unchanged. The `start` method computes and stores a Boolean `self.blocked` indicating whether the smallest sonar reading is less than a threshold, and then calls the parent's `start` method. The `step` method is similar. Finally, the `done` method returns `True` if either the parent `done` method returns `True`, or if one of the sonar readings is too short.

```
class XYDriverOA (XYDriver):
    def start(self, (sonars, pose)):
        self.blocked = min(sonars) < blockedThreshold
        XYDriver.start(self, (sonars, pose))

    def step(self, (sonars, pose)):
        self.blocked = min(sonars) < blockedThreshold
        return XYDriver.step(self, (sonars, pose))

    def done(self):
        return XYDriver.done(self) or self.blocked
```

Question 4: Let's try out the code; open the `XYDriverBrain.py` file in Idle.

a. Where it says `Your code here`, create a behavior, using `XYDriverOA` that moves the robot one meter in the `x` direction, or until a collision is detected. Load this file as a brain in Soar, run it in the `She` world.

b. Replace that command with one that gets the robot to do a dance at the current location (see the `dance` function in `XYDriver.py`). Load this file as a brain in Soar, run it in the `She` world.

HGBrain.py

We've defined a basic terminating state machine that drives directly to a location. Now, we need a way of using the planner to decide what locations to drive to. The control structure we want to have, at the top level, is that it calls the planner to make a plan, it executes the first macro-action in the plan and then it updates the current location and calls the planner again. This is kind of like a sequence of terminating state machines, but with the slight complication that don't know exactly what sequence of state machines we want to execute in advance.

So, we will define a new kind of terminating state machine class, called `IPE`, which stands for "interleaved planning and execution," that will drive to a goal by repeatedly planning and executing the first macro-action in the plan.

At initialization time, we just provide the desired goal, in grid indices, a grid world, and a planner for that grid world.

```
class IPE():
    def __init__(self, goalIndices, gw, planner):
        self.goalIndices = goalIndices
        self.gw = gw
        self.planner = planner
```

The `start` method will be called when this behavior is started, but also any time it is necessary to make a new plan (e.g., when a macro-action of moving from one grid square to another terminates).

We start by converting the current actual pose of the robot into grid indices, and then ask the planner to make a plan from the current indices to the goal indices, and draw the result. If it returns a plan of length greater than 1 (meaning that it actually has a step in it that needs to be done), then we take the first action (which will be something like `(1, 0)`, which means to move one grid square in the `x` direction, relative to the current pose), and convert it to a pose offset in real world coordinates. Then we make a new instance of the `XYDriverOA` behavior with the goal of making that relative motion. If there is no plan, or a plan exists but has length one, we will decide to do the `Wait` behavior, which will wait a few steps and try to make a new plan in case something has changed. Finally, the behavior for the macro action (either to make a relative motion or to wait), is started.

```
    def start(self, input):
        (sonars, (x, y, theta)) = input
        # Get grid indices for robot's x,y location
        self.currentIndices = self.gw.poseToIndices((x,y))
        # Make a plan
        plan = self.planner.plan(initialState = self.currentIndices,
                                 goalTest = lambda x: x == self.goalIndices)
        if plan:
            drawPlan(self.gw, plan)
        else:
            # Couldn't get there:  draw start and goal
            drawPlan(self.gw, [(None, self.currentIndices),
                               (None, self.goalIndices)])

        if plan and len(plan) > 1:
            # Return a terminating behavior to take the first action
            # plan[1] is the first real step,  [0] is the action part
            motion = self.gw.indicesToPose(plan[1][0])
            self.macroAction = XYDriverOA(motion, robot.goalEpsilon)
        else:
```

```
            # Wait 5 steps and see if we've been kidnapped
            self.macroAction = Wait(5)

        # Start the macroAction
        self.macroAction.start(input)
```

To step this whole behavior, we first check to see whether the current macroaction is done. If it is, then we call the `start` method, which will replan and initiate a new macroaction. If it is not yet done, then we simply step the current macroaction.

```
    def step(self, input):
        (sonars, (x, y, theta)) = input
        if self.macroAction.done():
            self.start(input)
        self.currentIndices = self.gw.poseToIndices((x,y))
        return self.macroAction.step(input)

    def currentOutput(self):
        return self.macroAction.currentOutput()
```

Finally, we're done when the robot is currently in the grid square that is its goal (note that it is not required, for example, to reach the center of the square; as soon as it enters the desired square, it is deemed to have reached the goal.

```
    def done(self):
        return self.currentIndices == self.goalIndices
```

Now, almost all the work is done. We just have to write the actual robot brain methods. We're leaving out some of the code to set parameter values, here; see the file for details. We make a `Map` from a set of boxes, and then make a `GridWorld` from that map. And then we make a window and draw a picture of the map in the window.

After that bookkeeping stuff, we make an instance of the `IPE` (interleaved planning and execution) class, with goal indices corresponding to real-world coordinates (3.5, 0.5) (near the bottom right corner of the maze), and the `GridWorld` and `Planner` we just made. Finally, we start that behavior, based on newly collected sensor values.

```
def setup():
    gw = D2GridWorldFromMap(Map(sheBoxes), xmin, xmax, ymin, ymax, gridN)
    gwPlanner = Planner(gw)
    robot.w = DrawingWindow(windowWidth, windowHeight,
                            xmin-0.5, xmax+0.5, ymin-0.5, ymax+0.5, "Plan")
    gw.drawWorld(robot.w, windowWidth)

    robot.planExecute = IPE(gw.poseToIndices((3.5, 0.5)), gw, gwPlanner)
    robot.planExecute.start(collectSensors())
```

The step method of the robot brain is now very short. If the `IPE` behavior is done, then the robot stops; otherwise, it reads the sensors, feeds the values into the `IPE` behavior, and executes the motor commands that the behavior generates. In this case, the motor commands are pairs of forward and rotational velocities, not discrete choices like `move` and `stop`.

```
def step():
    if robot.planExecute.done():
        doAction((0.0, 0.0))
    else:
        doAction(robot.planExecute.step(collectSensors()))
```

```
def collectSensors():
    return [sonarDistances(), cheatPose()]
def doAction(a):
    (fvel, rvel) = a
    motorOutput(fvel, rvel)
```

The only other interesting thing here is that, because we are running in simulation, and only studying the planning aspects of robot control, we are just cheating and getting the robot's true pose, in absolute simulator coordinates.

Question 5: Run the simulation again, this time picking a different (reasonable) goal for the robot, and change `HGBrain.py` to go there.

Question 6: Try changing `gridN` to 10 and to 40 and explain what happens in each case, and why.

Question 7: Currently, the `Planner` uses breadth-first search with dynamic programming. Change it to use depth-first search with dynamic programming. What happens? Why?

Checkpoint 1: 45 Minutes

- Be sure you have read the whole handout
- Explain effects of changing grid size and search method.

A more general goal condition

Currently, `HGBrain` is set up so that there's a single goal pose for the robot.

Question 8: Change the code so that it can accept a more general goal condition, such as a set of grid indices, or ranges on `i, j` indices. If your strategy for doing this breaks the code that draws the start and goal states when the planner fails, you can just not draw anything under those circumstances.

Checkpoint 2: 90 minutes

- Show that you can change the goal from a single state to a set of states, so that the robot would be satisfied reaching any of the states in the set. Demonstrate it in simulation.

Sequential goals

In the final project, we will be getting the robots to find a light in a maze; we'll give you a set of potential locations that might contain the light. So, you'll have to get the robot to go looking in each of those locations.

Question 9: Change the brain in `HGBrain.py` so that it commands the robot to go to three different goals in sequence. You shouldn't need to change anything about `IPE`. You might want to use the `SequentialTSM` class, a variant of which we've used before, and which is defined in `TSMFull.py`.

Question 10: In the file `XYDriver.py`, you'll find a procedure `dance`, which returns a terminating behavior that does a little robot dance. Make your robot do a victory dance every time it reaches one of its goals. Make the dance cooler, if you want to.

Checkpoint 3: End of lab

- Show that your program can make the robot go to several goals in sequence. And do the electric slide.

Post-lab Homework

Question 11: The system is using breadth-first search, with dynamic programming. What is the depth of the search tree when it finds a solution to the first planning problem it solved (before you changed the goal condition)? What is the typical branching factor (answer this *very* approximately)? Approximately (just get the order of magnitude right) how many nodes will be visited to find a solution with dynamic programming turned on? How about with dynamic programming off? Explain why.

Question 12: Given that we have `XYDriver`, why do we need the planner? What would happen if we just asked it to drive directly to the final goal location?

Exploration

Catching the bus!

This is worth 5 out of 10 exploration points. Use the code in the `lab11/exploration` *directory.*

Consider the problem of meeting up with another robot that is following a predetermined path with a known schedule: think of it as trying to catch a bus.

We want to construct a plan that will get us to be at some grid coordinates at the same time as the bus will be there. If we want to do that, we have to reformulate the state and action spaces that we have been using. Let's consider a new formulation of the state space, to include our robot's current grid coordinates as well as the current time.

You can experiment with this model in the simulator. Go to SoaR and pick `busWorld` (in the `lab11` folder) as your simulated world. The red square is the bus. Now just click on the **joystick** and **start**. The bus will sit at its initial position for a bit, then drive to another corner of the world, wait a while, drive to the next corner, and so on. In our model, the bus can only be caught when it is at a stop.

Go to the file `BusBrainShell.py`. The `busSchedule` function provided in that file will take a time as input and tell you where (in grid indices) the bus will be. An important thing to be aware of is that we are planning at a high level of abstraction as if every robot action takes the same amount of time, even though this isn't actually the case. We set `timePerRobotStep` to be 1.0 seconds. Use this value as the predicted amount of time that passes during each action.

One way to get insight into what is happening with the bus schedule is to write a simple Python loop that prints the bus location at a series of times, incrementing time by `timePerRobotStep`. Recall that a robot action is a move between adjacent grid squares and is modeled as taking `timePerRobotStep` time to execute. If the robot has caught the bus at one location, and the bus moves forward first, can the robot catch up to the bus again at the next step? If not, think about the next possible opportunity to catch the bus.

Exploration 1: Make the robot plan to catch the bus, and execute its plans, replanning dynamically whenever a high-level step ends.

Go to the file `BusBrainShell.py`. You'll find that there are three places where you might want to put some code. Location `#1` is where you would put things you need to do only once, like define a function or create an instance of an object. Location `#2` is where you'd put things you need to do every time the robot needs to replan (ultimately a call to the search function needs to go here, either directly or via a call to the `plan` method of a `Planner` object). Location `#3` is where you take whatever plan structure you have and hand the sequence of robot locations, of bus locations, and of times to the `drawUsAndBus` routine, and then extract out the first action for execution. Show the additions you made to the code and explain them.

Explain the behavior that you see. Does it make sense? Try it with the the robot able to make diagonal moves versus not. How does that change the behavior?

Exploration 2: Watching the planner run and printing out the starting state each time it is called, you should be able to see the time taken by actual robot moves. How long is the longest? The shortest? Try experimenting with values of `timePerRobotStep` that are both larger and smaller than the current setting. Explain what happens and why.

Update the map

5 points

Exploration 3: Change the system so that if you encounter an obstacle in a square that should be free, you change the map. Try starting with an empty map, and see if you can build a good map of the world.